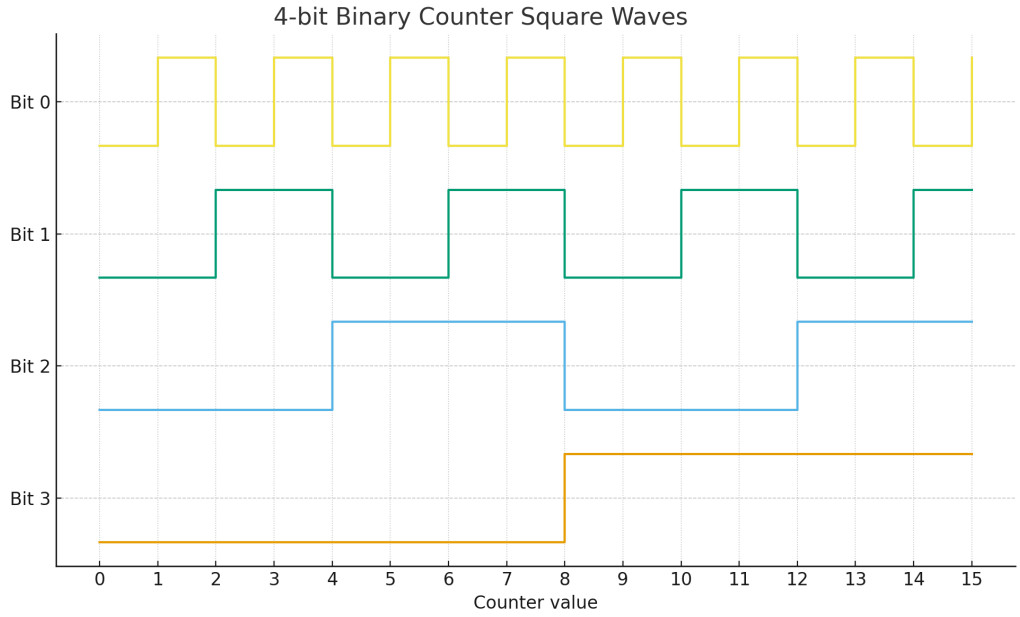
1. Walsh Functions

Think of a Walsh function as digital versions of sine and cosine waves.
Instead of smooth oscillations, they jump only between +1 and –1.

The Walsh functions can be constructed by multiplying together various square waves embedded in an integer counter.

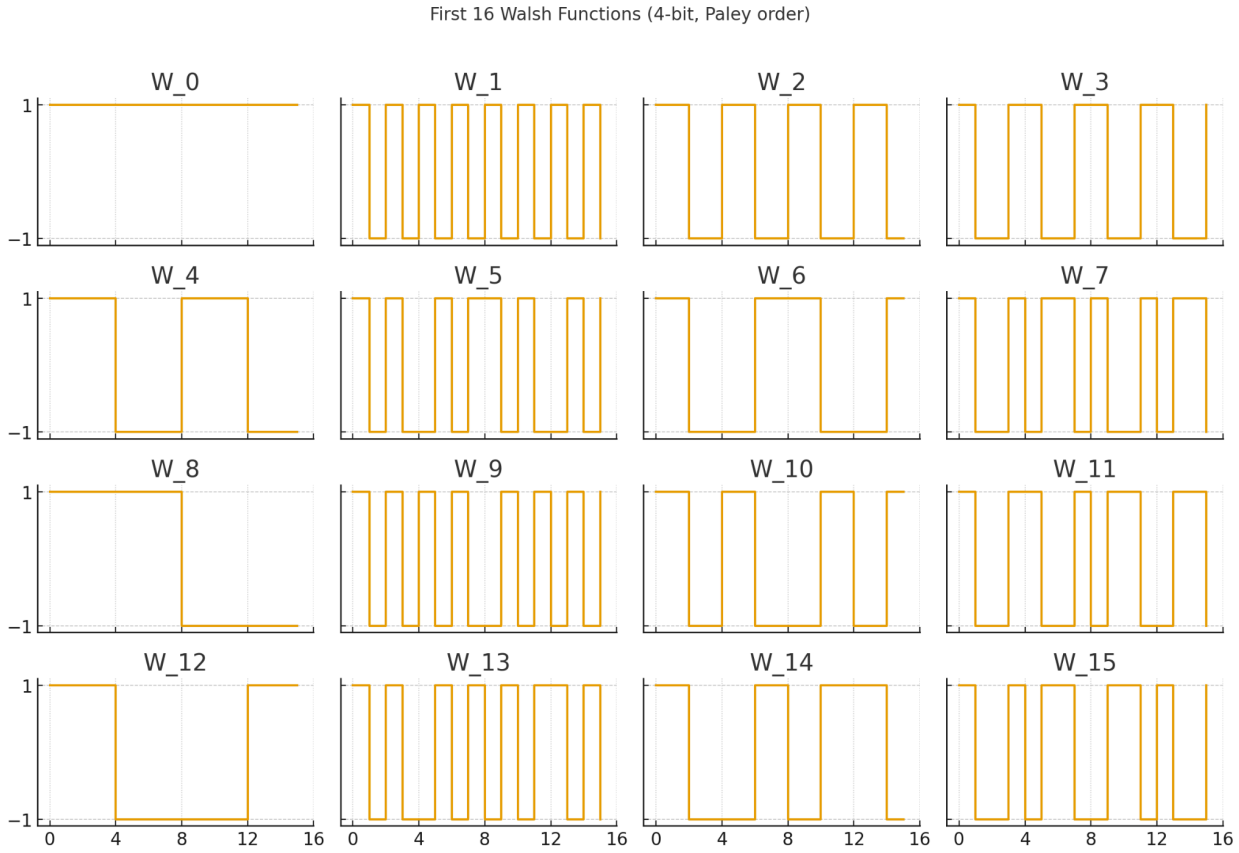


For example a 4 bit counter c embeds a fast changing square wave at bit zero (the least significant bit) and a slow changing square wave at bit 3 (the most significant bit.)

- How do we multiply them?
    - Write an integer `k` in binary: for example `k = 13` is `1101`.
    - Each binary digit tells you whether to include a certain square wave whose speed doubles at every step.
    - If the bit is 1, include that square wave; if 0, skip it.
    - Multiply together the chosen square waves.
    - Since bit multiply is the xor (exclusive or) function you can xor the chosen bits together as a quick multiply.

- The parity function is the xor of all the bits in an integer and is available as a CPU instruction or flag in most CPUs. Then you just logical ‘And’ `k` with the counter and take the parity to get a particular Walsh function.
- For some CPUs a better option is the popcount instruction (eg. Integer.bitCount(x) in Java) which counts the number of set bits in an integer, then logical ‘And’ that with 1 to get the parity function.
- Finally you map the 0,1 product (parity) to -1,1.

Because multiplying two ±1 signals is just another ±1 signal, you always end up with a ±1 pattern.



The Walsh functions can be ordered in a number of different ways. In the above diagram the Paley (k mask) order is given.

Another useful way to order them is by sequency (the number of transitions.) For example when `k = 6 (W_6)` the sequency is 4.

Orthogonality means that if you average the product of any two different Walsh functions over one full period, the positives and negatives cancel exactly to zero. That makes the Walsh Functions an orthogonal basis.

Key Advantages of Walsh Function

- **Binary hardware simplicity** – generated by counters and XORs.
- **Strict orthogonality** – clean channel separation and predictable interference rejection for digital signal processing.
- **Spectral flatness** – higher-sequency sequences mimic white noise while remaining deterministic.

In short:
Walsh functions are prized wherever you need *orthogonal, deterministic, square-wave like signals* for modulation, sensing, or hardware testing—making them valuable from optical networks to robotics labs, even when you never perform a Walsh–Hadamard transform at all.

2. Thue–Morse Sequence in Plain Language

The Thue–Morse sequence is built from the **parity of the number of 1-bits** in a binary counter.

- Count upward: 0, 1, 2, 3, 4 …
- For each number, write its binary form and count the 1s.
- If the count is even, output +1; if odd, output –1.

Example:

```
n:    0 1 2 3 4 5 6 7
bin:  0 1 10 11 100 101 110 111
ones: 0 1 1 2 1   2   2   3
seq: +1 −1 −1 +1 −1 +1 +1 −1
```

That infinite pattern is the Thue–Morse sequence.
It happens to be exactly the Walsh function whose binary mask has all bits set to 1 (all the embedded square waves are chosen and multiplied.)

---

3. 2's-Complement Counter and Parity

A **2's-complement counter** is just an ordinary binary counter that wraps around after its word size.
For an `m`-bit counter:

- It holds integers from 0 up to 2^m–1.
- When you add 1 past the top, it wraps back to 0 automatically.

The Thue–Morse sequence comes from:

```
popcount(n)  -> number of 1 bits
parity(n)    -> popcount(n) mod 2
output       -> +1 if parity even, −1 if parity odd
```

---

4. Permuted Thue–Morse via Constant Increments

Now, instead of stepping the counter by **1**, step it by a fixed **odd** constant `a` each time and wrap around using 2's-complement arithmetic:

```
x₀ = seed      (seed often 0)
x₁ = x₀ + a    (mod 2^m)
x₂ = x₁ + a    (mod 2^m)
...
```

This is exactly an **additive Weyl sequence** in computer arithmetic.
Because `a` is odd, this still visits all 2^m counter states, however in a permuted order.

For each value `x_k`, compute the parity of its bits:

```
output_k = +1 if popcount(x_k) even, −1 if odd.
```

Why it Works

- Adding an odd constant is a **one-to-one permutation** of the integers modulo 2^m.
- You still visit every binary word once before repeating, but **not in natural order**.
- Taking the parity after this permutation gives a **scrambled Thue–Morse sequence**.

Effectively you have taken the standard Thue–Morse pattern and **shuffled the index positions** without changing the set of values.
This destroys simple alignment with any single Walsh function while preserving the flat, noise-like spectrum that makes Thue–Morse useful.

---

5. Applications of the Permuted Version

- **Signal dithering / test signals**
  – The permuted sequence keeps the wide, nearly white Walsh–Hadamard spectrum but avoids correlations with the basic Walsh basis.
- **Pseudo-random generators**
  – Very cheap in hardware: just an adder and a parity (multi-input XOR) gate.
- **Spread-spectrum or scrambling**
  – Gives deterministic but hard-to-predict patterns without heavy computation.
- When combined with the fast Walsh Hadamard transform provides quasi-random projections with low computational cost convolution-like behavior.

Code (Java):

```
public class Walsh{

	// counter x, mask k

	public static int walsh(int x,int k){

		return 1-2*(Integer.bitCount(x&k)&1);

	}

	public static void main(String[] args){

		int seq=5;

		int k=seqToMask(seq,3);

		int seqM=maskToSeq(k,3);

		System.out.println("seq: "+seq+" mask: "+k+" seq from mask: "+seqM);
```

```
		for(int i=0;i<8;i++){
			System.out.println(walsh(i,k));
		}
		System.out.println();
		System.out.println("Permuted Thue–Morse sequence (constrained to 4 bits)");
		System.out.println("4 bit counter (0b1111) ,counter increment 5");
		for(int i=0,counter=0;i<16;i++){
			System.out.println(walsh(counter,0b1111));
			counter+=5; //add an odd number
		}

		System.out.println();
		System.out.println("Permuted Thue–Morse sequence (constrained to 32 bits (size of int))");
		System.out.println("counter increment 21, but not balanced +1's,-1's in general, more random.");
		for(int i=0,counter=0;i<16;i++){
			System.out.println(1-2*(Integer.bitCount(counter)&1));
			counter+=21; //add an odd number
		}

	}
	// sequency seq, number of counter bits nBits
	public static int seqToMask(int seq,int nBits){
		return reverseBits(seq^(seq>>>1),nBits);
	}
	// mask, number of counter bits nBits
	public static int maskToSeq(int mask,int nBits){
		int rev=reverseBits(mask,nBits);
		return grayCodeInverse(rev);
	}
```

```
        public static int reverseBits(int x,int nBits){
                return Integer.reverse(x)>>>(32-nBits);
        }

        public static int grayCodeInverse(int a) {
    a^=a>>>1;
    a^=a>>>2;
    a^=a>>>4;
    a^=a>>>8;
    a^=a>>>16;
    return a;
  }
}
```